SICS T92:01

Objektorientering i SICStus Prolog

Thomas Sjöland
SICS

december 1991

SICS Technical Report ISSN 1100-3154
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, SWEDEN

Objektorientering i SICStus Prolog

Thomas Sjöland
SICS, Box 1263, S-164 28 Kista, Sweden
tel: +46 8 752 1542
datorpost: thomas@sics.se

Sammanfattning

Denna tekniska utredning har gjorts på uppdrag av projektet "Applied logic programming" . Ininiativtagare var Seif Haridi, SICS, Mats Carlsson, SICS, och Hans Nilsson, Ellemtel. Syftet var att ange hur SICStus Prolog kan utrustas med faciliteter á la Prolog++. Texten nedan kan användas som diskussionsunderlag tillsammans med andra dokument om objektorientering och Prolog. Den kan även utgöra (del av) ett underlag för en eventuell implementering av stöd för objektorienterad programmering i SICStus Prolog. Implementationsförslagen är principiella. För att täcka hela Prolog++ måste givetvis alla inbyggda predikat ur Prolog++-manualen understödjas, liksom översättningen av Prolog++ till Prolog.

Bakgrund

Kombinationen av objektorienterad programmering och logikprogrammering har traditionellt inte behandlats i teorin för logikprogram. Den pragmatiska fördelen av objektorienterad programmering har dock gjort att några leverantörer av logikprogrammeringssystem har konstruerat objektorienterade stödsystem till prologsystem. Kommersiella system som är kända för oss är Prolog++ från Logic Programming Associates [Vasey, Spencer et.al.90], Logical Object Systems (LOS) från Imperial College (F. G. McCabe) [McCabe89] och ELSA från ELSA Software. Av dessa har vi tillgång till Prolog++. I den mån det påpekas skillnader i de tre systemen är beskrivningen av dessa hämtad ur LPAs dokumentation. Dessutom finns icke-kommersiella system som t.ex. LOCO/KIWIS som kopplar samman en databas, ett grafiskt användargränssnitt (Openlook) och en objektorienterad utvidgning av Prolog [Vermeir et.al. 89,91a,91b]. Arbete pågår på universitetet i Antwerpen med att anpassa LOCO/KIWIS till SICStus Prolog. LOCO/KIWIS är resultatet av ett ESPRIT-projekt där electruminstitutet SISU har deltagit [KIWIS91]. Systemet Explore/L [Fromherz91] är ett system som på många sätt liknar Prolog++. Rapporten som anges i referenslistan är på 72 sidor och innehåller ingående tekniska detaljer för en prologprogrammerare som vill implementera detta system. Även frågor om deklarativ semantik för objektorienterad logikprogrammering behandlas.

Allmänt

Allmänt kan sägas att kombinationen av objektorienterad programmering och logikprogrammering med Prolog har inslag av "ad hoc"-lösning. Det saknas "programmeringskultur" runt objektorienterad prologprogrammering. Detta leder liksom i andra fall, som t.ex. grafiska användargränssnitt (GUI), till att olika leverantörer kommer fram med sinsemellan inkompatibla lösningar. Vi lutar oss i denna framställning mot Prolog++ av flera skäl:

- SICS har kontakt med LPA.
- LPA delägs numera av Quintus Systems, ett Intergraph-företag vars Prolog-system i stor utsträckning tjänat som förlaga för designen av SICStus Prolog.
- Prolog++ är en genomarbetad produkt som är väl integrerad i ett marknadsledande prologsystem på Macintosh och PC-datorer.

System för objektorienterad programmering implementeras ofta tillsammans med en grafisk programmeringsomgivning. I denna finns förutom naturliga metoder att ge tillämpningarna ett grafiskt gränssnitt även interaktiva grafiska verktyg för att utveckla program. LPA saluför t.ex. MacObject som är en miljö för att utveckla Prolog++-program som integrerats i den grafiska miljön som LPA också utvecklat för Macintosh respektive Windows.

Logikprogrammering (med SICStus Prolog)

Logikprogrammering med SICStus Prolog [CaWiJAnSAnBoNiSj91,AlAnFlFrNiSu91] innebär kortfattat (och förenklat) att man programmerar definitioner av relationer (predikat). Två former av definitioner finns:

- *atomära klausuler* som definierar enkla fakta, t.ex:

```
farfar(nils,lisa).
far(sture,bengt).
far(bengt,anna).
```

- *villkorliga klausuler* där symbolen ':-' läses "om" t.ex:

```
farfar(X,Y) :- far(X,Z), far(Z,Y).
```

Till vänster om symbolen ':-' står en datastruktur, en s.k. *atom*, vars *namn* - i detta fall `farfar` - benämner klausulen. Formeln till höger om ':-' kallas *villkorsdel*. Denna kan vara en *konjunktion* (uttryck med infix-symbolen ',') eller en *disjunktion* (uttryck med infix-symbolen ';') av atomer eller formler eller en *atom* som eventuellt innehåller argumentdata. Argumentdata utgörs av *datastrukturer* vilka inleds med liten bokstav eller inramas med enkla citattecken och *logiska variabler* som inleds med stor bokstav. (`nils`

och `'X'` är datastrukturer, `X` är en variabel). Man kan även använda *datastrukturer* med argument t.ex. `träd(Arg1,Arg2)` som argumentdata:

```
fint_träd(löv).
fint_träd(träd(löv1,träd(Gren, löv2))) :- fint_träd(Gren).
```

Datastrukturer tillåts ha olika antal argument. För att skilja dem åt beskrivs de genom angivande av argumenttalet (t.ex. `träd/2`). En datastruktur utan argument (t.ex. `data/0`) benämns *konstant*. Datastrukturer benämnes ofta 'objekt' i litteraturen om logikprogrammering, men i denna text undviker jag att använda detta begrepp för att inte blanda samman det med det helt annorlunda objektbegreppet som utvecklats inom objektorienterad programmering. En samling definitioner med samma namn och argumenttal, (t.ex. `fint_träd/1, farfar/2` eller `far/2` ovan) kallas för en *predikatdefinition*. En samling predikatdefinitioner utgör ett *program*.

Ett program kan fördelas på flera filer och kan förutom definitioner även innehålla *direktiv*, procedurer som exekveras vid inläsningen.

Tolkningar av Prolog-program, logiska och icke-logiska

Ett program kan läsas *logiskt*, som definitioner av relationer över datastrukturer eller *procedurellt*, som definitioner av procedurer vilka utföres i en ordning som bestäms av programmets syntax. Programkörningen kan ses som en sökning efter en lösning till ett problem som formuleras som ett *mål*, t.ex målet

```
?- farfar(Farfar, Barnbarnet).
```

startar en sökning efter värden till de *logiska variablerna* `Farfar` och `Barnbarnet` så att målet kan anses vara bevisat. Prologsystemet svarar om sökningen lyckas med värden på variablerna:

```
Farfar=nils
Barnbarnet=lisa
```

Om det finns alternativa lösningar kan man få dem också, t.ex.:

```
Farfar=sture
Barnbarnet=anna
```

Exekveringen kan fås att avvika från den givna sökordningen genom att man skriver in speciella mål, s.k. *kontrollprimitiver*. I prologprogram kan man manipulera programmet

under körning genom att lägga till eller ta bort definitioner. Detta motiveras av att man vill se programmet som en databas av fakta och regler där vissa regler kan påverka innehållet i databasen. Program som modifierar databasen saknar logisk läsning.

Moduler i SICStus Prolog

I SICStus Prolog finns ett modulsystem som ger programmeraren möjlighet att strukturera sina definitioner så att en viss inkapsling uppnås. Predikat ingår i filer som kan namnges som *moduler*. Predikat kan *exporteras* och *importeras* mellan modulerna för att göra dem direkt kända utanför den definierande modulen. Predikat kan även vidareexporteras. Alla predikat kan nås genom att man i målet anger deras modulnamn. Moduler kan också skapas 'dynamiskt', dvs. under exekveringen. Modulsystemet i SICStus Prolog kan användas för att efterlikna arvsmekanismer i objektorienterad programmering. 'Inkapsling' av predikatnamn för att uppnå dolda, 'lokala' definitioner är dock inte möjligt.

Funktionell programmering

En klass av programmeringsspråk som är nära besläktad med logikprogrammeringsspråk är de funktionella programspråken. Ett funktionellt program kan ses som en samling definitioner av *funktioner*. Exekveringen blir då att på ett målinriktat, algoritmiskt, sätt söka efter värdet på en funktion som definierats i termer av andra funktioner och inbyggda, s.k. elementära funktioner. I funktionell programmering finns högre ordningens funktioner, dvs. funktioner som kan operera på funktioner som fullvärdiga data, och abstraktion, t.ex. lambda-abstraktion, som viktiga metoder för programmering. Funktioner har *typer* som kan härledas automatiskt vid inläsningen av programmet. Programkörningen utgör en *evaluering* av *värdet* hos ett funktionsuttryck för givna indata. I logikprogram kan de flesta mekanismer från funktionell programmering modelleras genom att man kodar funktioner som relationer. Typsystemet hos ett funktionellt programmeringssystem har ingen direkt motsvarighet i Prolog. Man kan dock använda datastrukturer för att sätta "tag" på värden och uppnå effekten av en typkontroll hos funktioner. I funktionell programmering är lat evaluering ett väsentligt begrepp. Detta innebär att funktionsuttryck evalueras först när värdet behövs och inte enligt en enkel syntaktisk evalueringsprincip. Lat evaluering kan ofta efterliknas genom att använda 'coroutining' i SICStus Prolog.
Användningen av begreppet "funktionssymbol" i logikprogrammeringslitteraturen gäller givetvis inte evaluerbara funktioner i denna mening utan istället datastrukturer. Förklaringen till detta begrepp ligger i modellteorin för logikprogram som definitioner av predikat över objekt i termtolkningen (symbolen '=' tolkas som syntaktisk likhet).

Procedurorienterad programmering

De kanske mest spridda programmeringsspråken som C, ALGOL, ADA, FORTRAN, Cobol etc. baseras på idén om *procedurer*. Liksom för klausuler i logikprogrammering och funktionsuttryck i funktionell programmering kan man i *procedurorienterad programmering* definiera procedurer *rekursivt* . Ett *mål* delas upp i *delmål* som körs successivt. En procedur definierar hur ett antal *tillståndsvariabler* förändrar sina värden genom att programmet körs. Under exekveringen skapas rekursiva *instanser* av delmängder av tillståndsvariablerna för proceduren, s.k. *lokala omgivningar*. Procedurerna kan också påverka värdena hos tillståndsvariabler som nås från alla omgivningar, s.k. globala omgivningar. Överföring av värden mellan instanser av procedurer kan ske på ett flertal olika sätt. I språket C används kopiering av värden som princip. För att innehållet i en tillståndsvariabel skall kunna modifieras måste en *pekare* överföras till proceduren.

Objektorienterad programmering

På senare tid har *objektorienterad programmering* blivit populärt [Haridi91]. Med detta avses en programmeringsmetodik som utnyttjar en hierarki av objektdefinitioner och d:o instanser. Ett objektorienterat system använder ärvningsmekanismer och exekveringskontrollen kan beskrivas med sändning av meddelanden mellan objekt. Objekten är modifierbara och innehåller *tillståndsvariabler (attribut)*, *evaluerbara funktioner* och *metoder*. Det förväntas i avancerade objektorienterade system att objekt kan skapas och tas bort genom utförande av programsatser (sändning av meddelanden) samt att systemet kan automatiskt lösa arvskonflikter (*skuggning, multipel ärvning*). Objektsystemet bör även kunna skydda innehållet i objekten för icke-önskvärd åtkomst och/eller modifikation, samt stödja programmering av s.k. *demoner*, dvs. kod som exekveras när vissa händelser inträffar.

Definitioner

I rapporten "Jämförelse mellan Ada och C++" [Sahlin90], citeras definitioner ur boken "Research Directions in Object-Oriented Programming" [Shriver87], s. 508.

Objekt Ett objekt har en mängd operationer och ett tillstånd som minns effekten av operationerna.

Klass En klass specificerar ett gränssnitt för operationer. Den fungerar som en mall efter vilken objekt med det givna gränssnittet kan skapas.

Klassärvning Klassärvning är en mekanism för sammansättning av gränssnitt från en eller flera superklasser med gränssnittet från en subklass. En subklass kan komma åt operationerna i en superklass om det inte finns några namnkonflikter.

Sahlin skriver vidare:

> Språk med stöd för objekt kallas i boken för "object-based", med stöd för objekt och klasser kallas "class-based" och språk med objekt, klasser och klassärvning kallas "object-oriented". Enligt denna definition blir Ada "object-based" och C++ "object-oriented".

LPAs terminologi är något annorlunda. Normalt används i texter om objektorientering begreppet "klass" för att tala om beskrivningen av ett gränssnitt och "objekt" för att beskriva de instanser som kan skapas under exekveringen. I denna text används LPAs konvention att tala om "objekt" för klass och "instanser" för objekt. Objekt och instanser är i stort sett likvärdiga men med den väsentliga skillnaden att instanserna inte syns explicit i programkoden utan skapas under exekveringen.

Objektorientering i logikprogrammering

Objektorientering har i logikprogrammeringssystem implementerats på några principiellt olika sätt:

- genom att man betraktar predikatdefinitionerna som objektdefinitioner, modifierar begreppet bindningsomgivning med olika inkapslings- och åtkomstmekanismer och implementation av arvsrelationer, m.m. I ESPRIT-projektet ALPES [ALPES90] valdes en liknande lösning kallad "contextual logic programming" som kan vara av intresse att studera. Denna infallsvinkel övervägs dock inte vidare i denna text.

- genom att man betraktar ett objekt som en samling av predikat, vilka definierar *attribut* (objektvariabler), *evaluerbara funktioner* och *metoder* för det namngivna objektet. I systemet införs mekanismer för att skapa objekt och instanser samt för *sändning* av meddelanden till objekten (instanserna) och för datadriven exekvering av metoder. Prologsystemet ligger till grund för implementationen av objektsystemet genom att Prolog++-programmen översätts till prologprogram. Prologprogram kan användas från objektens metoder och funktioner.

Objekten kan ses ha denna form:

```
objekt(<ObjektNamn>,<InstansNr>,<Attribut>,<Funktioner>,<Metoder>).
```

Refererade komponenter (attribut, funktioner och metoder) söks via hierarkilänkarna om de inte finns i det aktuella objektet. (Hierarkilänkarna är attribut). Komponenter kan även ärvas från namngivna objekt utan hänsyn taget till hierarkin genom att detta speciellt anges. Demoner är metoder som anropas då vissa händelser inträffar, t.ex. att ett attribut eller en instans skapas, tas bort eller modifieras.

Statiska och dynamiska objektsystem

Ett *statiskt* objektsystem skapar inga objekt under körningen. Därför bortfaller behovet att särskilja objekt och instanser helt. Varje objekt som används har sin egen definition och en explicit angiven mängd av tillståndsvariabler. Användbarheten hos ett statiskt objektsystem är begränsad.

Ett *dynamiskt* objektsystem tillåter att programmeraren skapar objekt i systemet som instanser av de givna objektdefinitionerna. Genom att länkar mellan objekt anges kan metoder, attribut och funktioner från andra objekt utnyttjas utan att definitionerna upprepas.

LPAs Prolog++

Följande operatorer definieras för Prolog++.

```
:- op(1200,fy,[open_object,close_object]).
:- op(1150,fx,[private,dynamic]).
:- op(800,fx,['<-').
:- op(800,xfx,['<-']).
:- op(700,xfx,
[':=','+=','-=','*=','/=',':==','+==','-==','*==','/==']).
:- op(600,fx,inherit).
:- op(400,yfx,'//').
:- op(100,yfx,'::').
```

Inbyggda predikat introduceras i manualen för Prolog++.

Objektdefinitioner och objektinstanser

En *objektdefinition* beskriver ett *objekt* men även en klass av *objektinstanser*. När objektinstanser skapas används objektdefinitionen som mall. En ny uppsättning tillståndsvariabler allokeras och objektinstansen får ett namn som man kan använda vid sändning av meddelanden. I Prolog++ kan meddelanden sändas till både objekt och objektinstanser. Distinktionen mellan objekt och instanser är att instanser inte namnges i källkoden, utan instanserna och deras attribut får nya unika namn när de skapas. Ärvning används för att hitta komponenter när de saknas i definitionen av objektet som mottar meddelandet. En direkt arvslänk kan överflygla ärvningen via objekthierarkin. Nyckelordet `inherit` används för att ange detta.

Attribut

Ett objekt (en instans) har en uppsättning unika tillståndsvariabler som benämnes *attribut*. Dessa kan användas i uttryck och förändras med en uppsättning operationer.

Metoder

De *metoder* som tillhör ett objekt utförs när objektet mottar ett meddelande med begäran om exekvering. En metod förstås som en lokal procedur.

Evaluerbara funktioner

I Prolog++ finns möjligheten att definiera exekverbara funktioner. Dessa kan ingå i uttryck och är väsentligen metoder som returnerar ett värde.

Objekthierarkier och arvsmekanismer

När objekt saknar en komponent kan systemet leta efter denna i ett annat objekt. Genom att definiera vilka objekt som ligger högre formar man en partiell ordning av objekt, en *objekthierarki.* Ärvning innebär att komponenten söks i de klasser som anges som högre. *Multipel ärvning* uppnås genom att flera "föräldrar" tillåts.*Skuggning* innebär att komponenter som hittas i objekt som ligger lägre i hierarkin gör att komponenter från "högre" objekt inte syns. En separat ordning kan etableras genom att precisera *explicit ärvning* för attribut, metoder och/eller funktioner.

Inkapsling och abstraktion

I och med att alla objekt definieras inom parentes (open_object, close_object) och komponenterna översätts så att klausuler för attribut, funktioner och metoder anger objektnamnet explicit är komponenterna *inkapslade.* I Prolog++ kan komponenter dessutom definieras som *privata.* med nyckelordet `private`. Då blir dessa enbart kända inuti objektet självt. I övrigt kallas de *publika.* För publika komponenter behövs inget speciellt nyckelord.
Namnen på ett objekt och namnen på dess publika komponenter utgör dess *abstraktion.*

Operationer på objektdefinitioner och objektinstanser

Man kan skapa objektinstanser baserat på objektdefinitioner och även ta bort instanser. En instans anses normalt ligga en nivå under det objekt det skapas med avseende på. Om komponenter har definierats som `dynamic` kan de läggas till och tas bort från ett objekt (eller en instans).

Operationer på objektattribut

Värdena hos attributen kan användas och /eller modifieras. Till attributet liksom till objekten kan associeras en eller flera demoner.

Kommunikation och exekveringskontroll i Prolog++

Den centrala operationen för ett objektorienterat system är överföring av meddelanden. Meddelanden skickas till objekt och är av tre typer:

1) begäran om exekvering av en metod:

```
obj1 <- message(hej)
```

2) begäran om modifikation av ett attribut:

```
obj1::attribute3 := nytt_värde
```

3) I uttrycken kan även förekomma attributvärden och funktionsuttryck. Dessa kan också betraktas som överföring av meddelanden, ett för att hitta attributet, och ett som återsänder dess värde till den som skickat meddelandet för att användas i det uttryck vari attributet förekommer:

```
X=obj2::function(1,2,3)
```

I Prolog++ bestämmer exekveringsmekanismen ett unikt objekt som svarar för ett attribut, en metod eller en funktion. Eftersom Prolog++ baseras på Prolog tillåts alternativ som kan användas vid backtracking. Dessa söks dock enbart i det valda objektet även om det finns definitioner på andra ställen i objekthierarkin. Detta kallas för *skuggning* ("overriding inheritance"). I LOS finns två olika mekanismer för ärvning. Förutom skuggning finns en mekanism där unionen av alla metoder (funktioner, attribut) i hierarkin används för prologsökningen. Med den mekanismen måste hierarkin definieras annorlunda för att uppnå effekten av skuggning. I Prolog++ kan samma effekt uppnås genom att använda attributet `super/1` och inkludera en klausul i objektet för att distribuera meddelanden uppåt i hierarkin:

```
...
metod(InData) :-
        super(myself) <- metod(InData).
...
```

Genom att hierarkilänkarna är attribut kan explicita sökningar i hierarkin programmeras med stor frihet.

Polymorfism uppnås genom att samma metodnamn används för att definiera olika beteenden i olika objekt.

En grupp meddelanden kan skickas till en grupp av objekt som en operation.

Demoner

En *demon* är en metod som exekveras vid en speciell händelse i Prolog++ :

- när instanser av objekt skapas eller tas bort
- när nya värden definieras för attribut i objekt eller objektinstanser
- när den lokala databasen hos ett objekt eller en instans ändras (med detta avses när metoder eller funktioner ändras eller när attribut tas bort eller läggs till)

En demon definieras som ett objekt och kan ingå i objekthierarkin, sända meddelanden, anropa prologmål osv. En vanlig användning är för att fånga speciella händelser t.ex. för att generera grafisk utmatning.

Inbyggda objekt

I Prolog++ finns två inbyggda objekt, s.k. meta-objekt:

- `object` för att kontrollera användardefinierade objekt
- `instance` för att kontrollera run-time instanser av dessa objekt

Implementationsidéer för SICStus++

Implementationsförslagen nedan baseras på ett försök att förstå vad som sker bakom kulisserna i Prolog++ genom studium av manualen för Prolog++, ej på något dokument som beskriver LPAs implementation.

I valet av representation för de olika språkkomponenterna (objekt, instanser, attribut, metoder, funktioner, demoner, hierarki och andra länkar) måste implementatören avväga effektivitet mot arbetsinsatsen som erfordras för respektive implementationsstategi.

Implementationsstrategi

En tämligen omedelbar implementation av de flesta av mekanismerna i Prolog++ kan göras i SICStus Prolog genom att representera objektens komponenter som ett antal klausuler med objektnamn och instansnummer som parameter. Objektnamnet och instansnumret tillsammans med attributet `super` används för att nå rätt objektkomponent (metod, funktion, attribut).

Om SICStus Prolog utrustas med mekanismer för att effektivt skapa, modifiera och använda en mängd av tillståndsvariabler som en del av databasen kan Prolog++ implementeras mycket effektivt.Om man använder sekvensen `retract, assert` för att implementera tilldelning av attribut blir operationer på attribut inte lika effektiva.

Några väsentliga egenskaper bör upprätthållas av alla implementationer:

• Prologmål skall kunna anropas från metoder och funktioner i Prolog++-objekt.

• Språket Prolog++ skall kunna kompileras och dekompileras till/från Prolog. Detta ger möjlighet till utnyttjande av Prolog-kompilatorn.

Objekt

I Prolog++ används prologsyntaxen för definitioner.
En objektdefinition avgränsas av

```
open_object <ObjectName>.
...<definitioner av hierarki, attribut, metoder, funktioner>
close_object <ObjectName>.
```

Dessa definitioner kan översättas till klausuler med namn och instansnummer explicit angivet (objekt har `No`=0, instanser har `No`>0). T.ex.:

```
object(<ObjectName/No>).
super(<ObjectName/No>,<Lista av andra ObjectName/No>).
attribute(<ObjectName/No>, <AttributeName>, <AttributeValue>).
method(<ObjectName/No>, <MethodName>, <Method>).
function(<ObjectName/No>, <FunctionName>, <Function>).
```

Attribut

Attribut specificeras med `=/2` eller `is/2`, t.ex.:

```
...
vem = allan.
listan is [].
...
```

Dessa deklarationer kan översättas till prologklausulerna:

```
attribute(<ObjectName/No>, vem, allan).
attribute(<ObjectName/No>,listan, []).
```

`ObjectName` står för det objekt som attributdefinitionen förekommer inom och `No` anger en unik instans för att man skall kunna hålla reda på samma attributnamn i olika instanser.

Metoder

En metod specificeras t.ex.:

```
dothis(Attribute, NewValue) :- myself::Attribute := NewValue.
```

Här ser vi dels operationen "::" som söker efter attributet "Attribute" i objekthierarkin, dels operationen ":=" som modifierar attributets värde i objektdatabasen.

En översättning blir:

```
method(<ObjectName/No>,dothis(AttributeName,NewValue)) :-
        '::'(<ObjectName/No>,AttributeName,FoundAttribute),
        ':='(FoundAttribute,NewValue).
```

Evaluerbara funktioner

I Prolog++ finns evaluerbara funktioner som attribut, t.ex.:

```
square(X) = X*X :- X>0.
```

Funktionen översätts lämpligen (om man inte vill ha icke-deterministiska funktioner) till:

```
function(<ObjectName/No>,square(X),Sq) :- X>0, !, Sq is X*X.
```

Objekthierarkier och arvsmekanismer

Attributet `super` anger vilka objekt som ligger högre upp i hierarkin. Mekanismen för sändning och mottagning av meddelanden kan använda dessa objekt som kandidater för att hitta metoder, funktioner eller attribut som inte hittas i objektet självt.

Operationerna som kan läsa värden hos attribut löser upp objekthierarkin för att finna det aktuella attributet, medan operationer som modifierar ett attribut dessutom måste modifiera databasen i Prolog eller de globala tillståndsvariablerna.

Inkapsling

I Prolog++ kan man skriva

```
private.
```

i definitionen av ett objekt. De därpå följande definitionerna av komponenter blir då osynliga utanför objektet självt. Detta implementeras med ett extra argument i komponentklausulen. Endast publika komponenter kan nås vid sökningen i hierarkin.

Operationer på definitioner och instanser

Instanser skall kunna skapas baserat på en objektdefinition. En instans skall även kunna tas bort.

Operationer på attribut

Ett attribut skall kunna läggas till ett objekt (en instans) och även kunna tas bort. Värdet hos ett attribut skall kunna användas i uttryck och ändras genom tilldelning. För

identifieringen av attributet användes normalt arvshierarkin. De operationer som understöds i Prolog++ är `'::'/3` för att hitta värdet hos ett attribut, `':='/2` för tilldelning, `'+='/2` för inkrementering, `'-='/2` för dekrementering, `'/='/2` för division av aktuellt värde, `'*='/2` för multiplikation med aktuellt värde. I samtliga fall kan en demon exekveras. Motsvarande operationer finns även utan väckning av demoner t.ex. `':=='/2`. De två klasserna av tilldelningsoperationer kallas 'noisy' respektive 'quiet'.

Kommunikation mellan objekt (instanser)
Meddelanden skall kunna skickas till objekt (instanser). "Multicast" skall understödjas så att flera meddelanden skall kunna skickas till flera objekt i en operation.

Demoner
Metoder skall kunna associeras till speciella händelser för ett objekt så att metoden automatiskt anropas när händelsen inträffar.

Sammanfattning och rekommendationer

Mekanismerna för Prolog++ kan implementeras med rimlig effektivitet om SICStus Prolog utvidgas med operationer för att allokera och modifiera globala tillståndsvariabler som innehåller godtyckliga prologtermer. Demoner kräver dessutom att prologmål kan associeras till tillståndsvariablerna. Explicit avallokering är nödvändig.

En enklare, men mindre effektiv implementation är möjlig genom att använda prologfakta för att representera attribut. Denna metod är adekvat om den huvudsakliga beräkningen ligger i metoder och funktioner genom anrop till Prolog.

En måhända besvärlig fråga är huruvida Prolog++ är det lämpligaste systemet att basera en objektorienterad utvidgning av SICStus Prolog på. En industriell utvärdering där man jämför Prolog++ eller Explore/L på Macintosh med t.ex. LOCO/SICStus som antas bli tillgänglig under våren 1992 rekommenderas innan slutgiltigt beslut tas. Rapporten om Explore/L verkar vara tillräckligt detaljerad för att kunna utgöra en implementationsspecifikation för ett examensarbete.

Referenser

[ALPES90] J.A.S. Allegria, A. Natali, N. Preston, C. Ruggieri, ALPES Final Report, Advanced Logic Programming Environments, (ESPRIT P973), CRIL Frankrike, DEIS, Univ. of Bologna, Italien, ENIDATA, Italien, LRI, Univ. of Paris-Orly och LSI, Univ. of Toulouse, Frankrike, Techn. Univ. of Munich, Tyskland, Universidade Nova de Lisboa, Portugal, 1990

[AlAnFlFrNiSu91] J. Almgren, S. Andersson, L. Flood, C. Frisk, H. Nilsson, J. Sundberg, SICStus Prolog Library Manual, SICS technical report T91:12B, september 1991

[BrFrHa89] Per Brand, Tommy Frisk, Seif Haridi, Industriell tillämpning av logikprogrammering, Swedish Institute of Computer Science (SICS) och Infologics AB i samarbete med Mekanförbundets stödkommitté, Mekanresultat 88002, April 1989

[CaWiJAnSAnBoNiSj91] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, T. Sjöland, SICStus Prolog User's Manual , SICS technical report T91:11B, september 1991

[Haridi91] S. Haridi, SICS, Expert Systems for Improved Crop Management, konsultuppdrag för FN, Egy/88/024, september 1991

[Fromherz91] M. P. J. Fromherz, Explore/L An Object-Oriented Logic Language, Institut für Informatik der Universität Zürich, Nr 91.06, juni 1991

[KIWIS91] M. Ahlsén, A D'Atri, P Johannesson, E. Laenens, N. Leone, P. Rullo, P. Rossi, F. Staes, L. Tarantino, L. Van Beirensdonck, F. van Cadsand, W. Van Santvliet, J. Vanslembrouck, B. Verdonk, D. Vermeir (ed.), The KIWIS Knowledge Base Management System, Rapport 91-23, Universitaire Instelling Antwerpen, Belgien, mars 1991

[McCabe89] F. G. McCabe, Logic and Objects, Ph.D. thesis, Dept. of Computing, Imperial College of Science and Technology, 1989

[Sahlin90] D. Sahlin, Jämförelse mellan Ada och C++, SICS (rapport till chefen för grundteknik på Televerket), 1990

[Shriver87] Eds. B.Shriver and P. Wegner, Research Directions in Object-Oriented Programming, MIT Press 1987

[Vasey, Spencer et.al.90] P. Vasey, C. Spencer, D. Westwood, A. Westwood, Prolog++, version 1.0 Programming Reference Manual, Logic Programming Associates Ltd., London, England, 1990

[Vermeir et.al.89] E Laenens, Philips Applications & Software Services, Eindhoven, Nederländerna, D. Vermeir, B. Verdonk, Dept. of Computer Science, Univ. of Antwerp, Belgien, LOCO, a Logic-based Language for Complex Objects. I *Proc. of ESPRIT'89 Technical Conf.*, Project 2424, pp.604-616 , 1989

[Vermeir et.al.91a] D. Vermeir, Dept. of Mathematics and Computer Science, Univ. of Antwerp, A simple interface from LOCO to X-Windows, rapport från ESPRIT project 2424, 1991

[Vermeir et.al.91b] E. Laenens, D. Vermeir, Univ. of Antwerp, Belgien, N. Leone, P. Rullo, CRAI Italien, Efficient query evaluation in a language combining object-oriented and logic programming, rapport från ESPRIT project 2424, 1991